QSYM : A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing

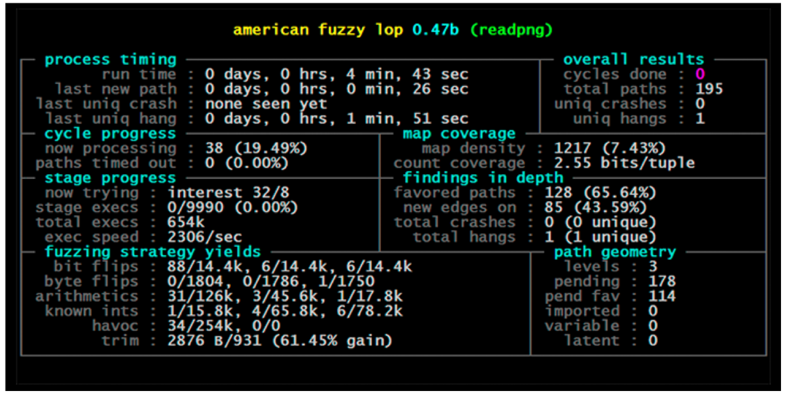
Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang †, and Taesoo Kim

Georgia Institute of Technology & Oregon State University †

27th USENIX Security Symposium

August 16, 2018

Two popular ways to find security bugs: Fuzzing & Concolic execution



Fuzzing

Symbolic Execution

Fuzzing and Concolic execution have their own pros and cons

- Fuzzing
  - Good: Finding general inputs
  - Bad: Finding specific inputs

- Concolic execution
  - Good: Finding specific inputs
  - Bad: State explosion

Hybrid fuzzing can address their problems

- Use both techniques: Fuzzing + Concolic execution

- Find specific inputs: Using concolic execution
- Limit state explosion: Only fork at branches that are hard to fuzzing

Hybrid fuzzing has achieved great success in small-scale study

- e.g.) Driller: a state-of-the-art hybrid fuzzer
  - Won 3rd place in CGC competition
  - Found 6 new crashes: cannot be found by fuzzing nor concolic execution

However, current hybrid fuzzing suffers from problems to scale to real-world applications

- Very slow to generate constraint
- Cannot support complete system calls
- Not effective in generating test cases

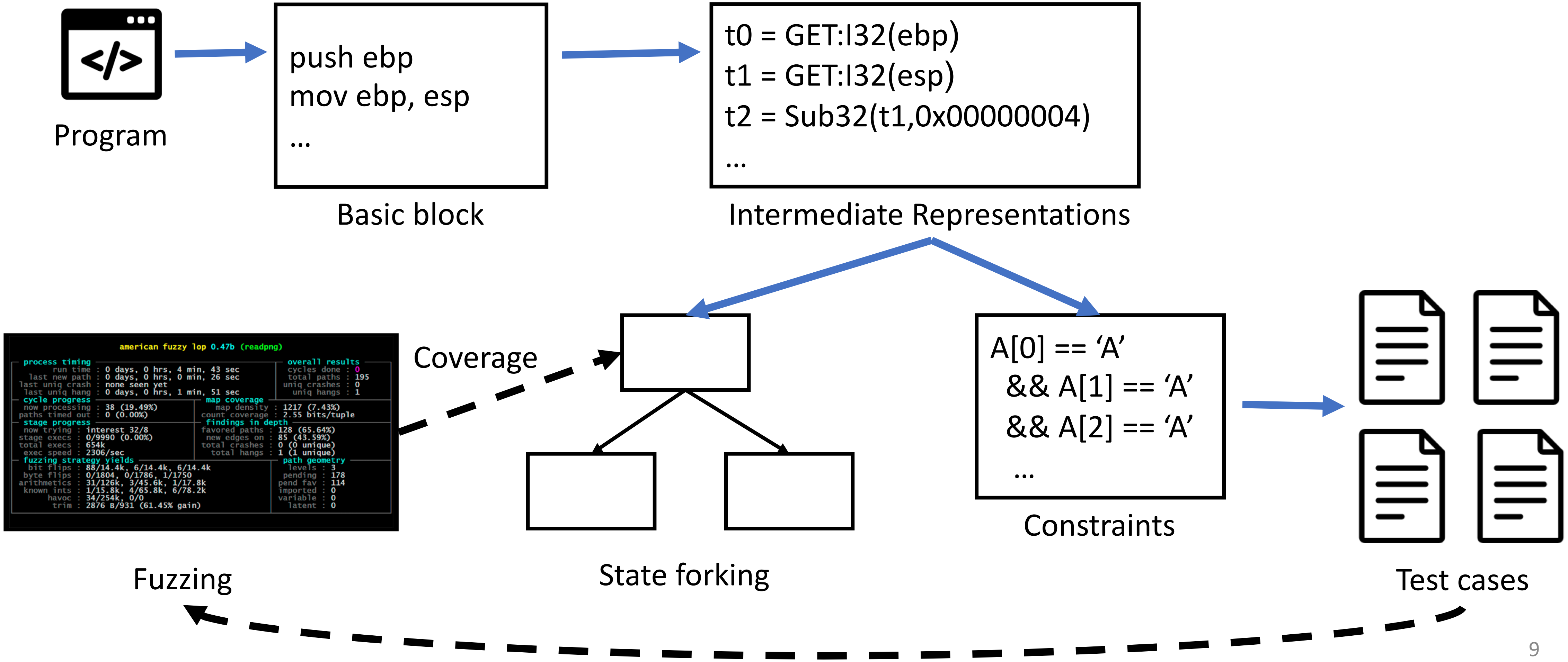
Our system, QSYM, addresses these issues by introducing several key ideas

- Discard intermediate layer for performance
- Use concrete environment to support system calls
- Introduce heuristics to effectively generate test cases

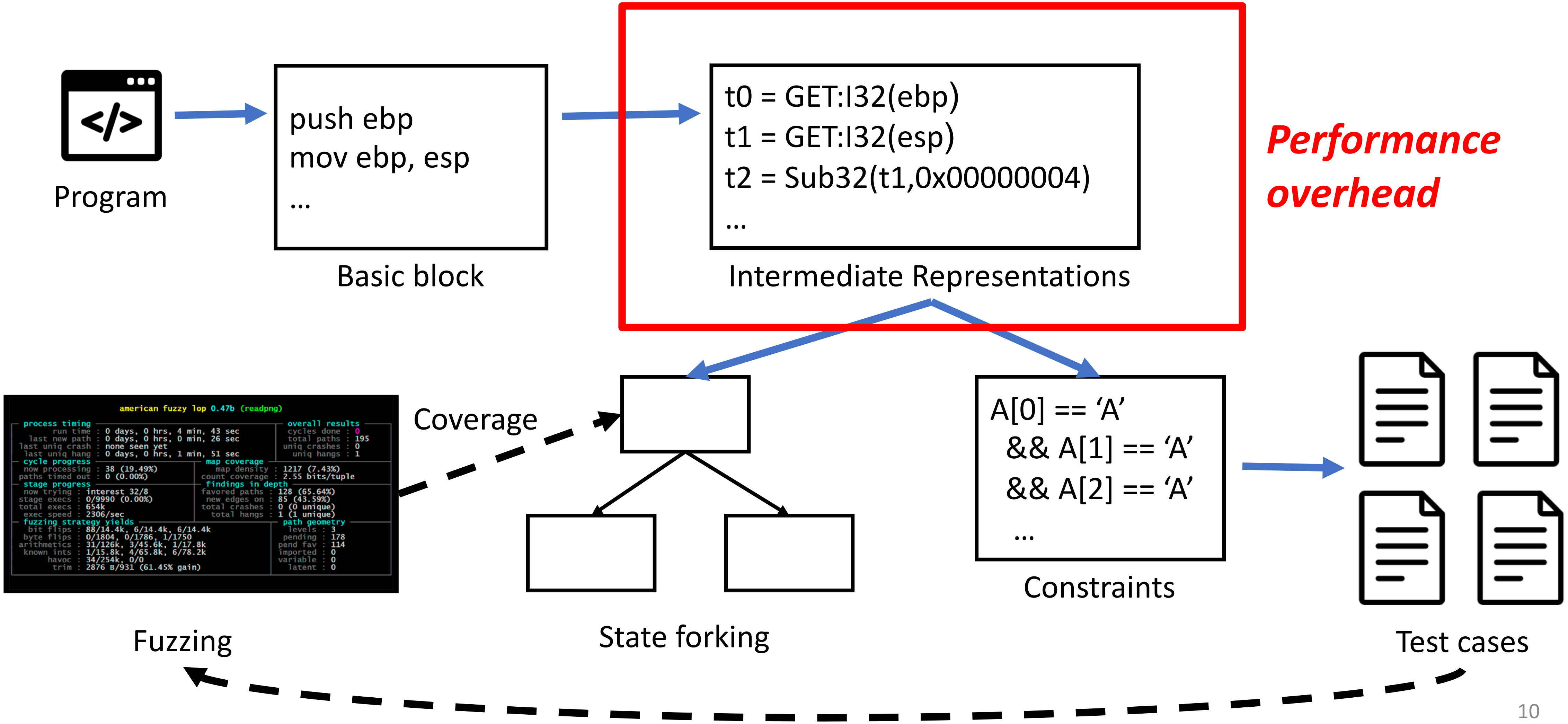
QSYM is scalable to real-world software

- 13 previously unknown bugs in open-source software
- All applications are already fuzzed (OSS-Fuzz, AFL, …)
  - Including ffmpeg that is fuzzed by OSS-Fuzz for ***2 years***
- Bugs are hard to pure fuzzing – require complex constraints

Overview: Hybrid fuzzing in general

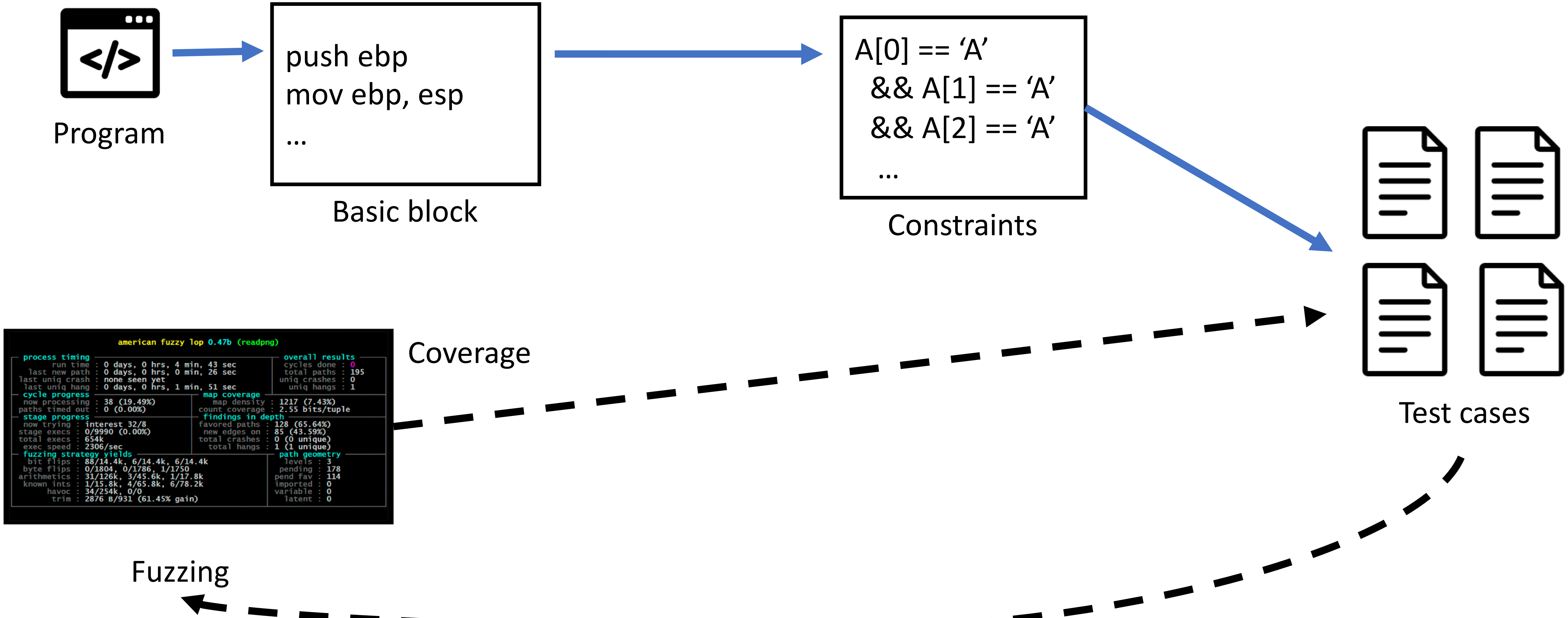


Overview: Hybrid fuzzing in general

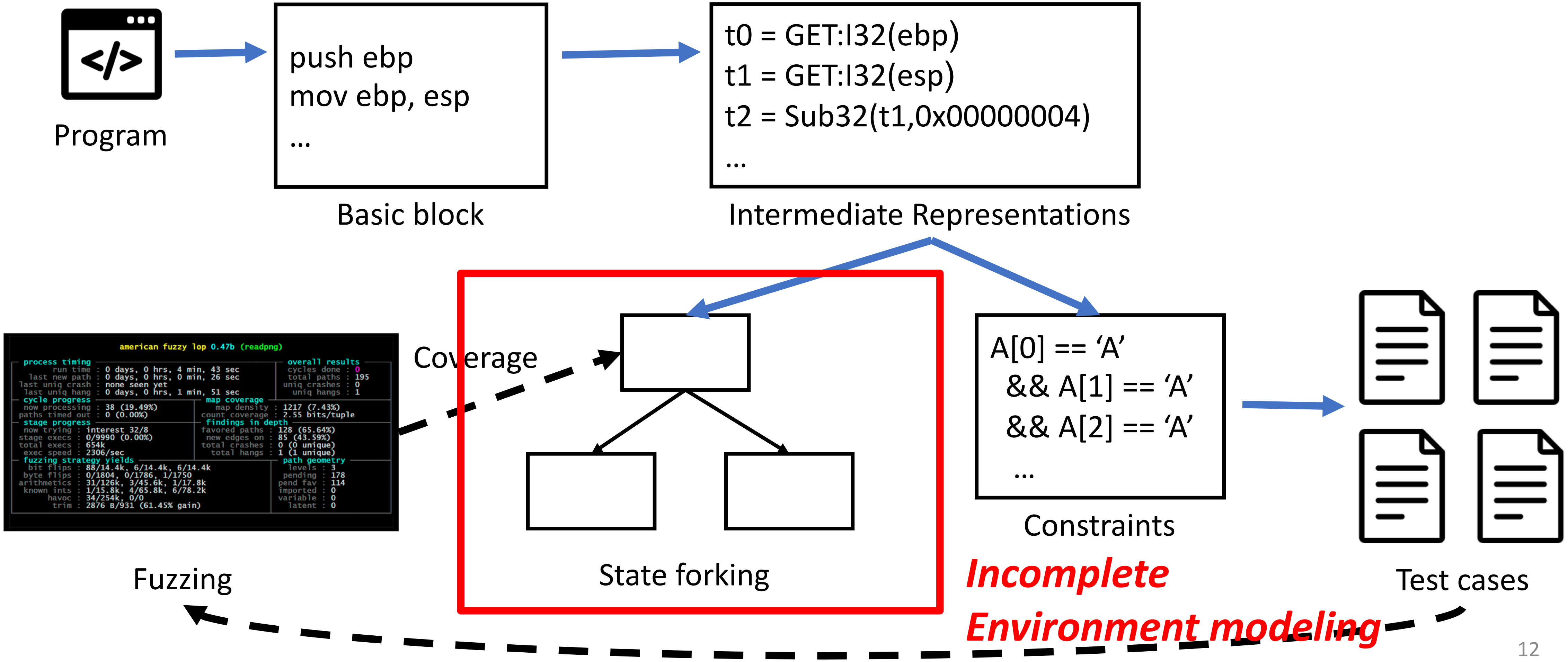


Overview: QSYM

1. Instruction-level execution

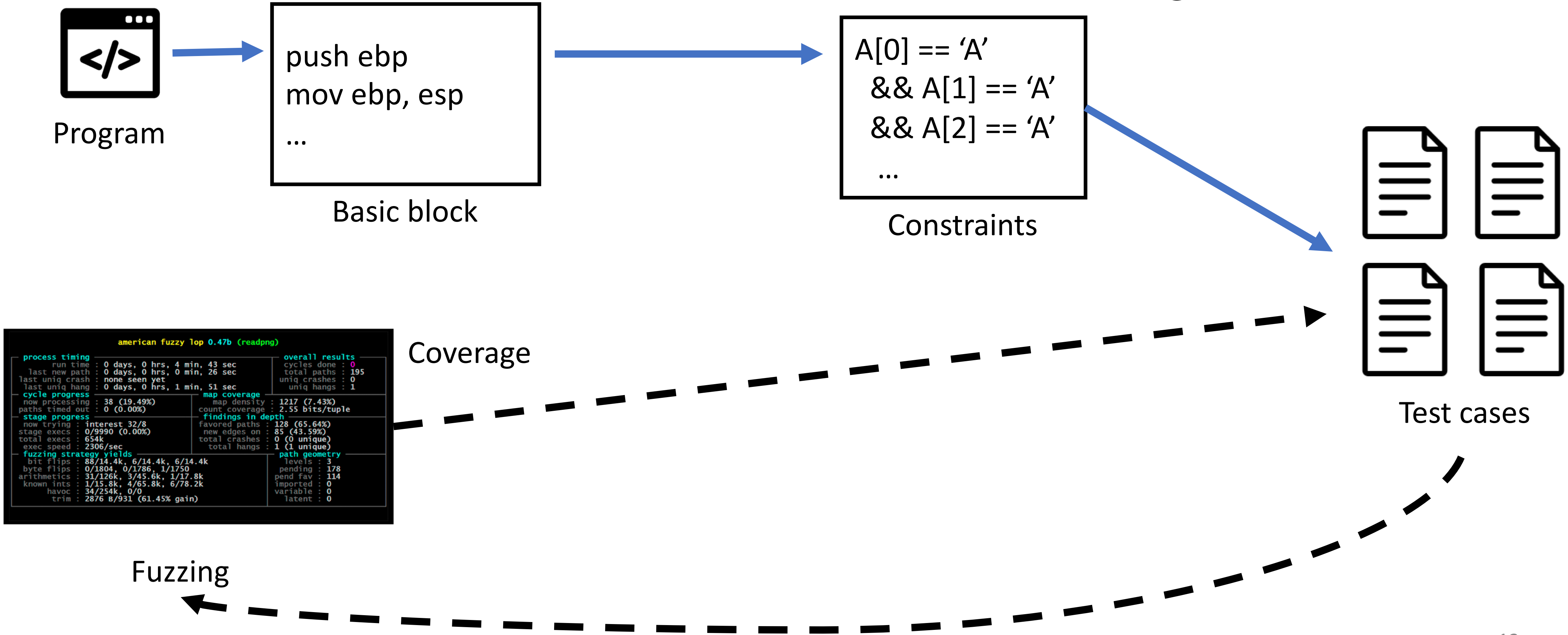


Overview: Hybrid fuzzing in general

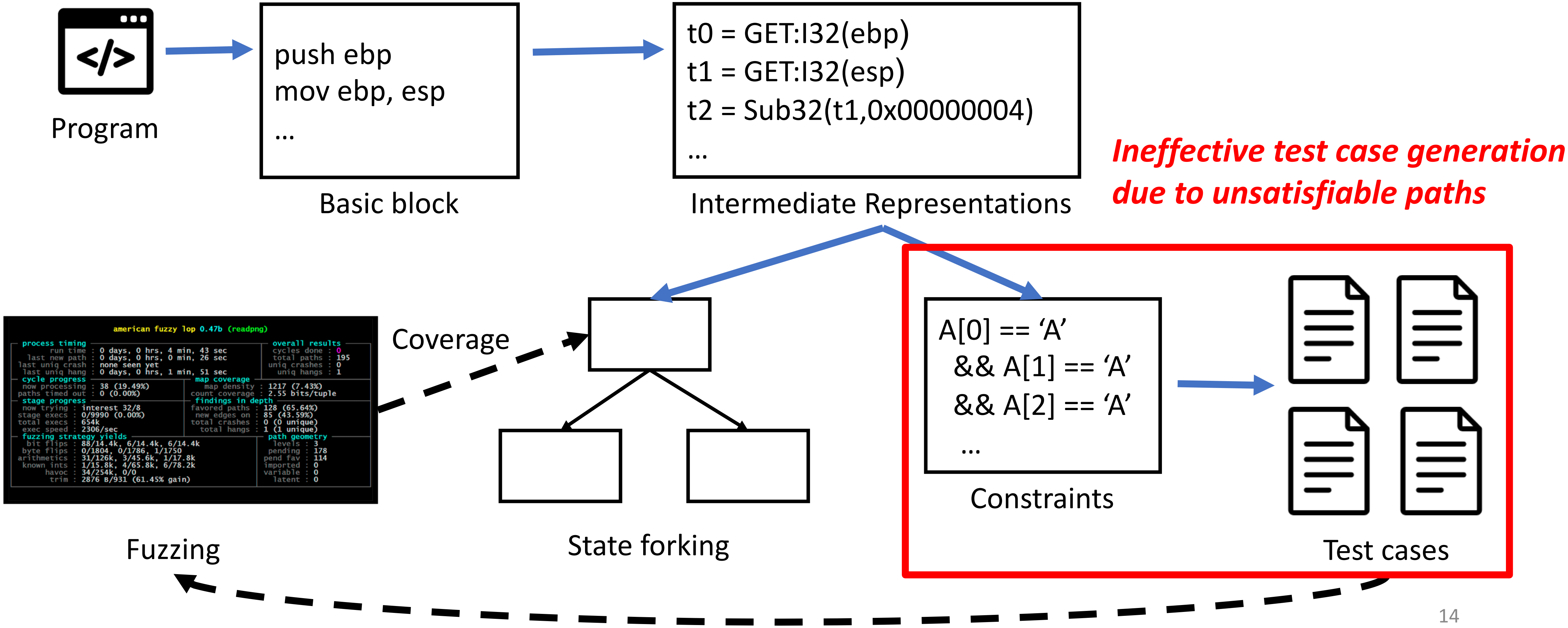


Overview: QSYM

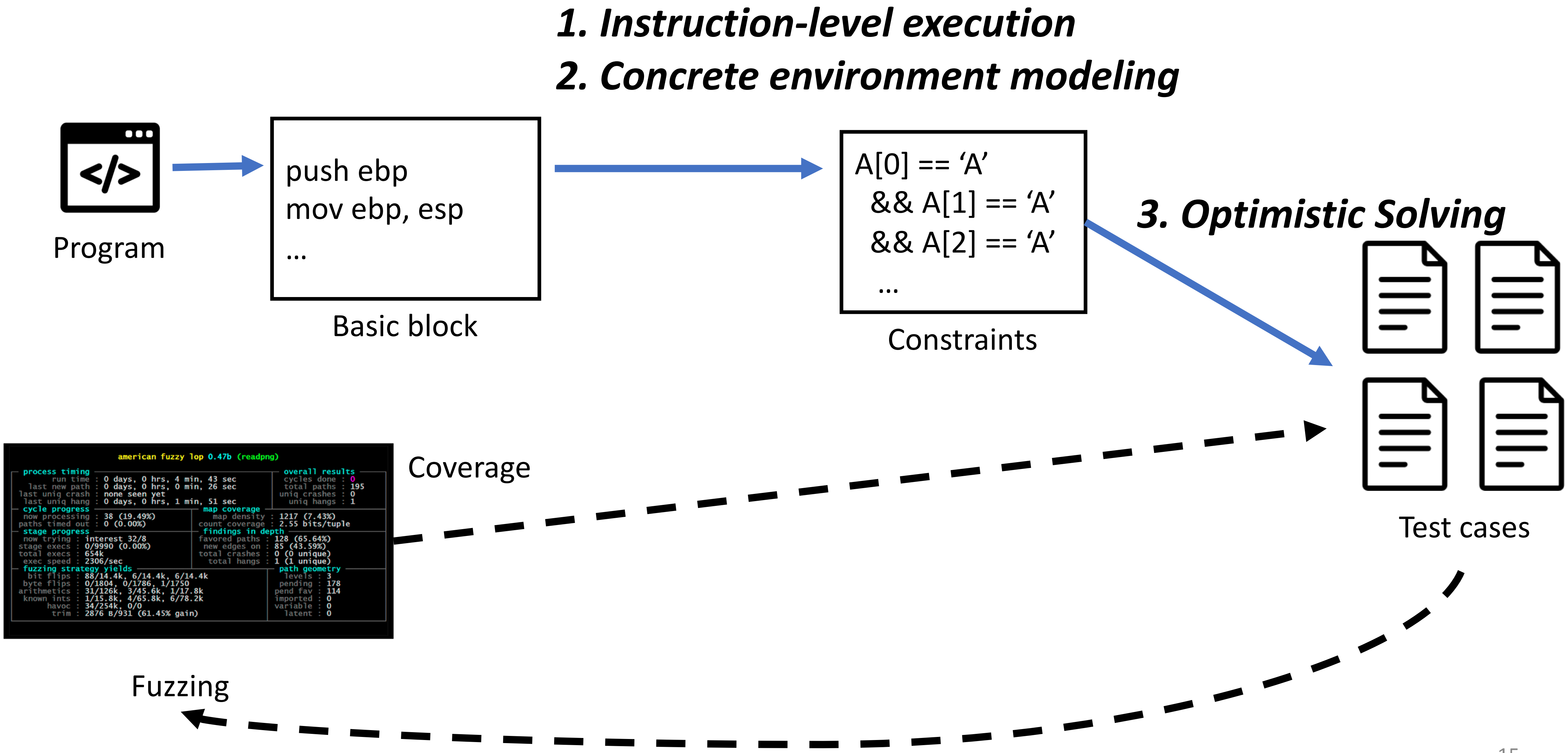
1. ***Instruction-level execution***
2. ***Concrete environment modeling***



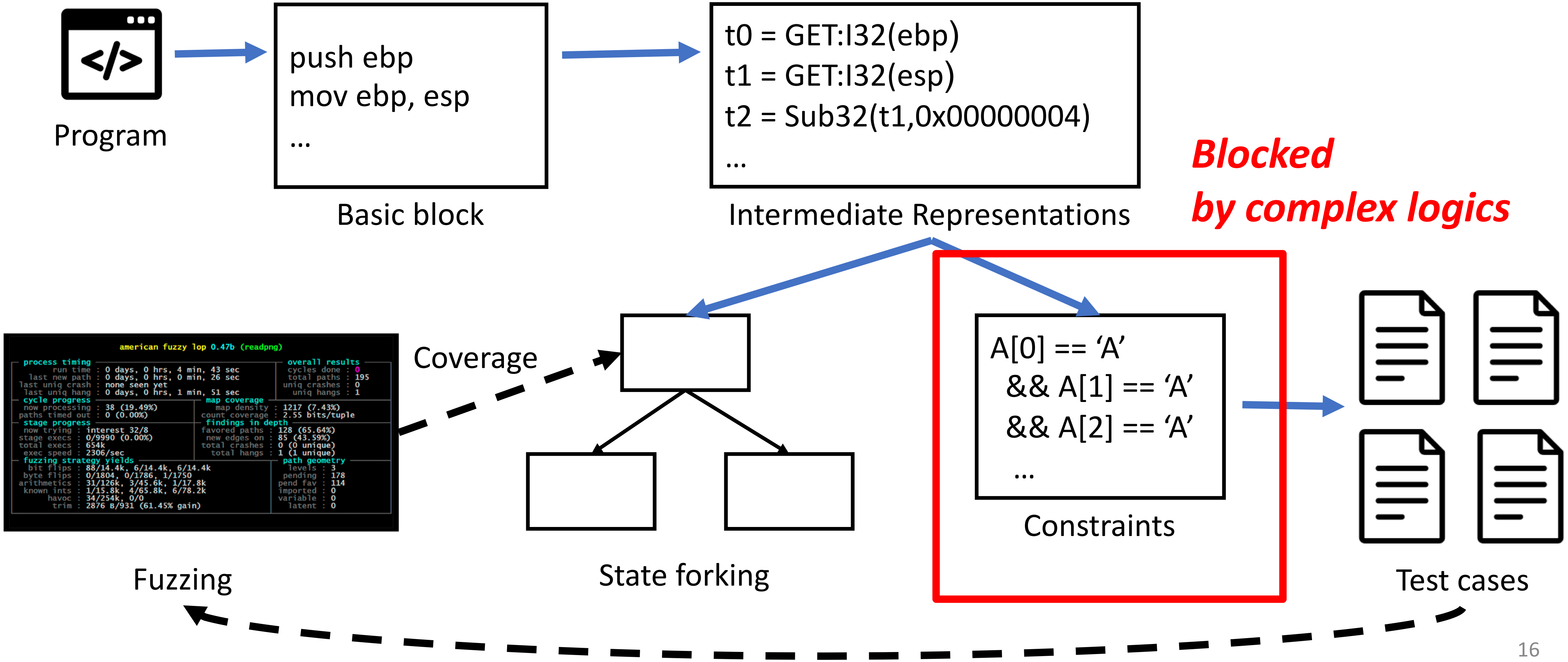
Overview: Hybrid fuzzing in general



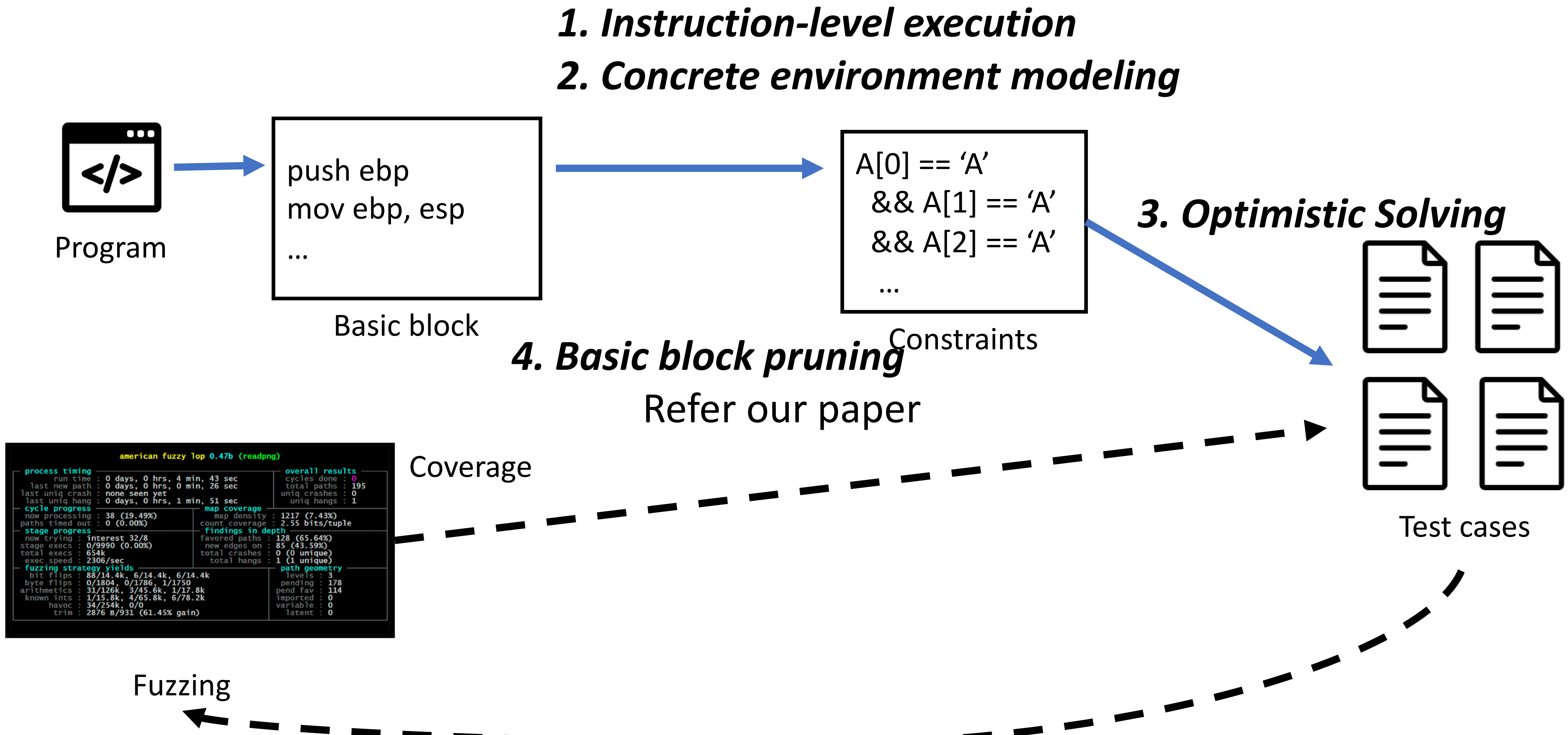
Overview: QSYM



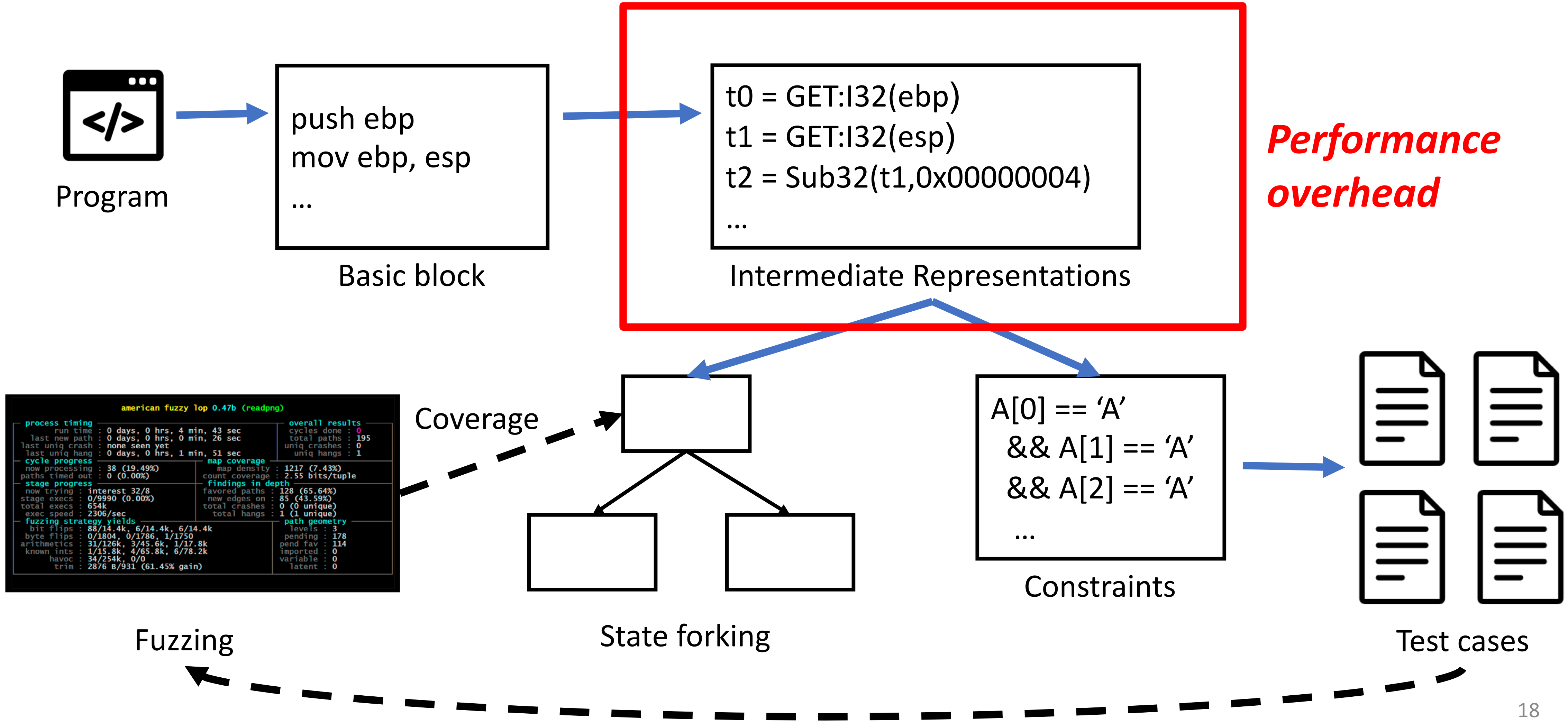
Overview: Hybrid fuzzing in general



Overview: QSYM



Overview: Hybrid fuzzing in general



Intermediate representations (IR) are good to make implementations easier

- Provide architecture-independent interpretations
- Can re-use code for all architectures
- e.g. angr works on many architectures: x86, arm, and mips

Problem1: IR incurs significant performance overhead

- Increase the number of instructions
  - 4.7 times in VEX (IR used by angr)

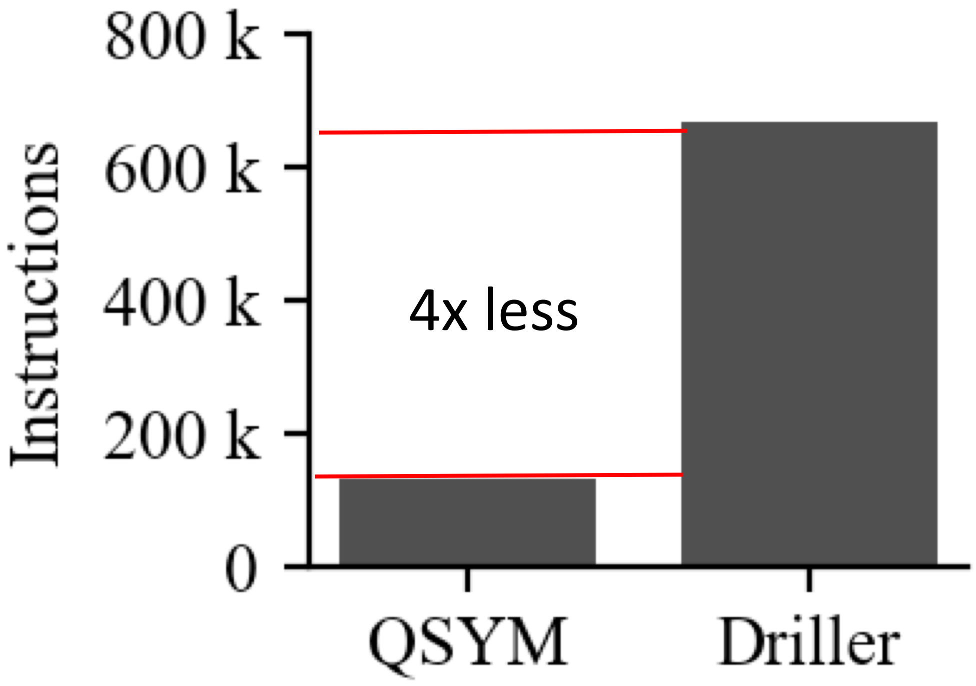
- Need to execute a whole basic block symbolically
  - Due to caching and optimization
  - Only 30% of instructions need to be symbolically executed

Solution1: Execute instructions directly without using intermediate layer

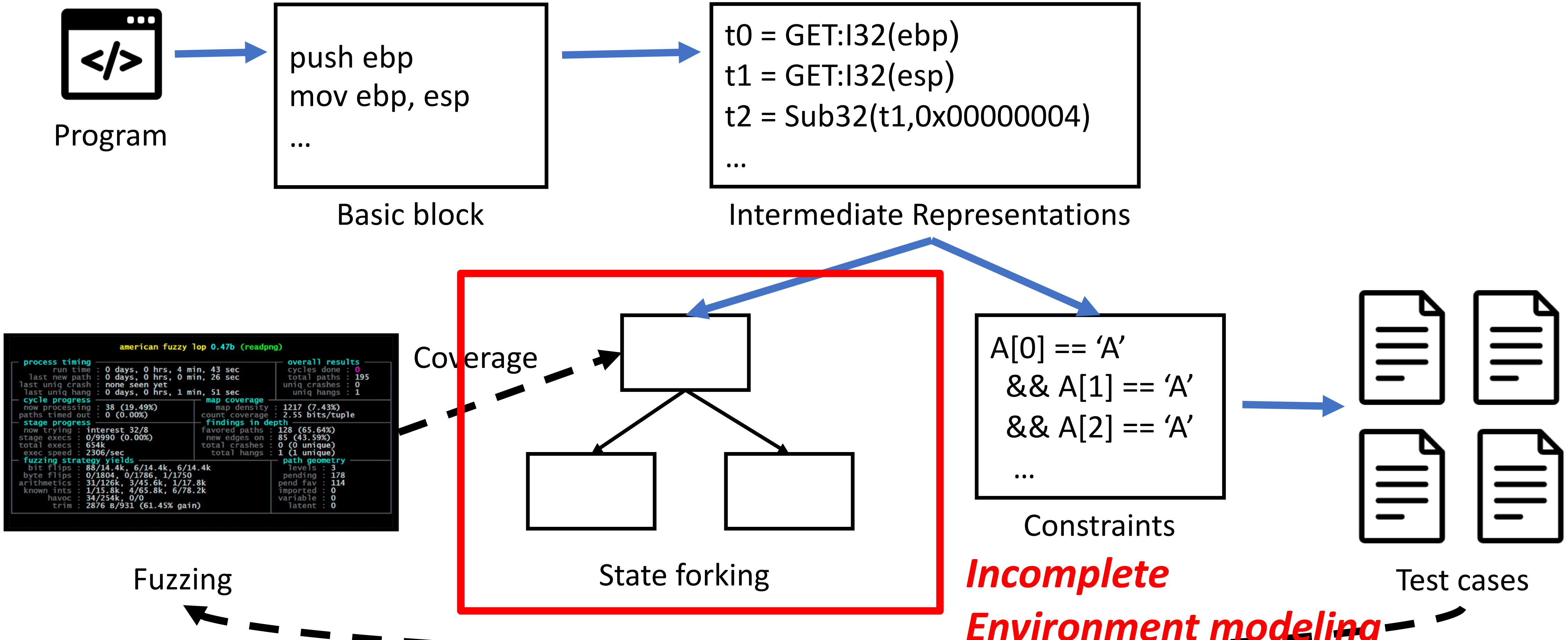
- Remove the IR translation layer
- Pay for the implementation complexity

QSYM reduces the number of instructions to execute symbolically

- 126 CGC binaries



Overview: Hybrid fuzzing in general



State forking can reduce re-execution overhead for constraint generation

- No need to re-execute to reach the state
  - Recover from the snapshot

State forking for kernel is non-trivial

- State in concolic execution = Program state + Kernel state

- Forking program state is trivial
  - Save application memory + register
  - Save constraints

- Forking kernel state is non-trivial
  - Need to maintain all kernel data structures
  - e.g., file system, network state, memory system …

Problem2: State forking introduces problems in either completeness or performance

- Kernel modeling
  - e.g.) angr
  - Pros: Small performance overhead
  - Cons: Incompleteness – angr supports only 22 system calls in Linux

- Full kernel emulation
  - e.g.) S2E
  - Pros: Completeness
  - Cons: Large performance overhead

Solution2: Re-execute to use concrete environment instead of kernel state forking

- Instead of state forking, re-execute from start

- High re-execution overhead
  - Instruction-level execution
  - Basic block pruning

  - Limit constraint solving: Based on coverage from fuzzing

Models minimal system calls and uses concrete values

- Only model system calls that are relevant to user interactions
  - e.g.) standard input, file read, …

- Other system calls: Call system call using concrete values
  - e.g.) mprotect(addr, ***sym_size***, PROT_R)
    → mprotect(addr, ***conc_size***, PROT_R)

Problem: Concrete environment results in incomplete constraints

- Add implicit constraints
  - e.g.) mprotect(addr, ***sym_size***, PROT_R)
    → mprotect(addr, ***conc_size***, PROT_R)

- Without knowing semantics of system calls
  - Concretize: Over-constrained
  - Ignore: Under-constrained

Unrelated constraint elimination can tolerate incomplete constraints

```
x = int(input())
y = int(input())

# Incomplete constraints
mprotect(addr, x, PROT_R)

if y * y == 1337 * 1337:
    bug()
```

Constraints for x (Incomplete)
&& y * y == 1337 * 1337

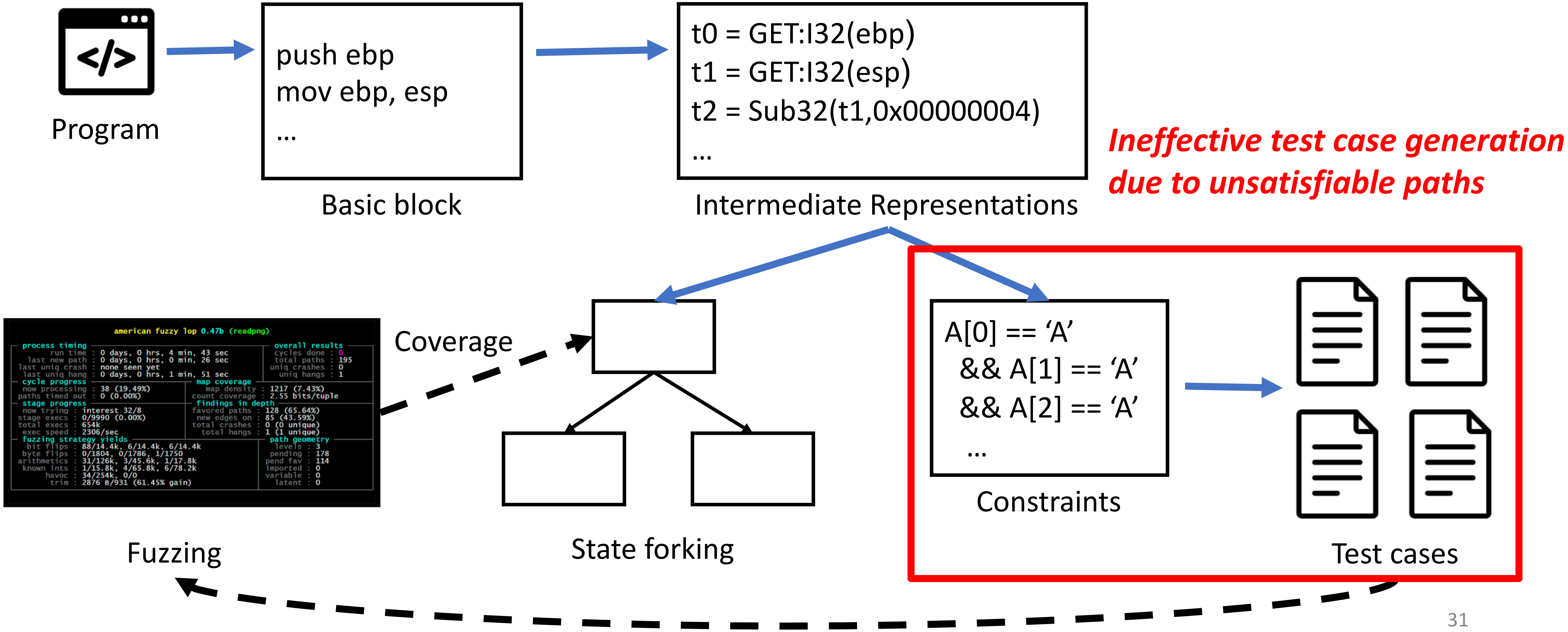
Path constraints

y * y == 1337 * 1337

Branch dependent constraints

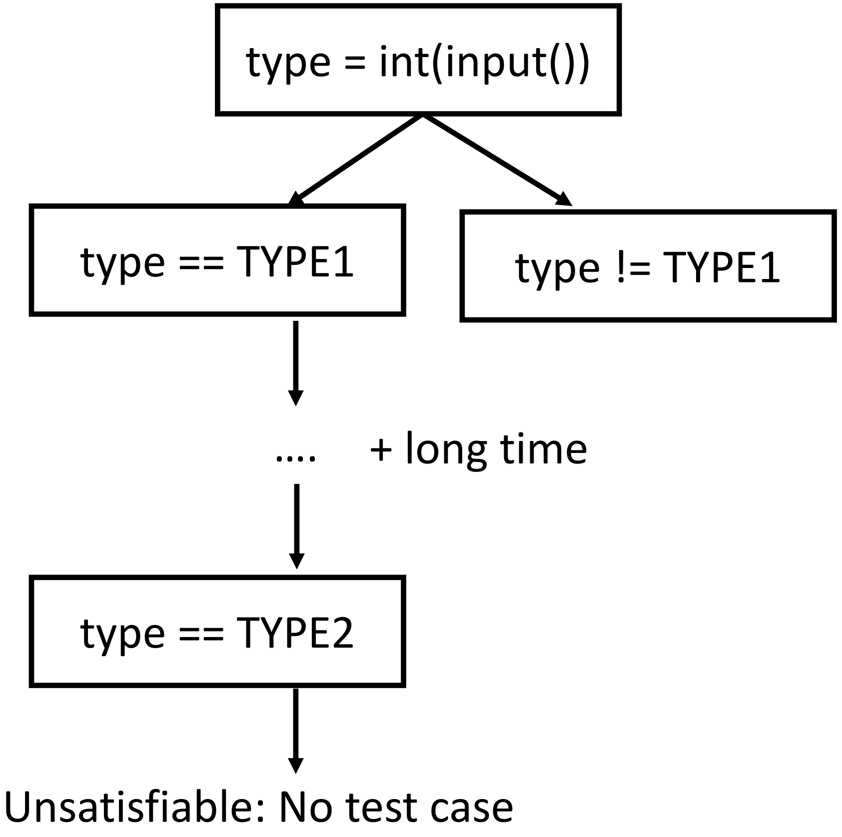
x = Use concrete value
y = 1337

Overview: Hybrid fuzzing in general



Problem3: Over-constrained paths results in no test cases

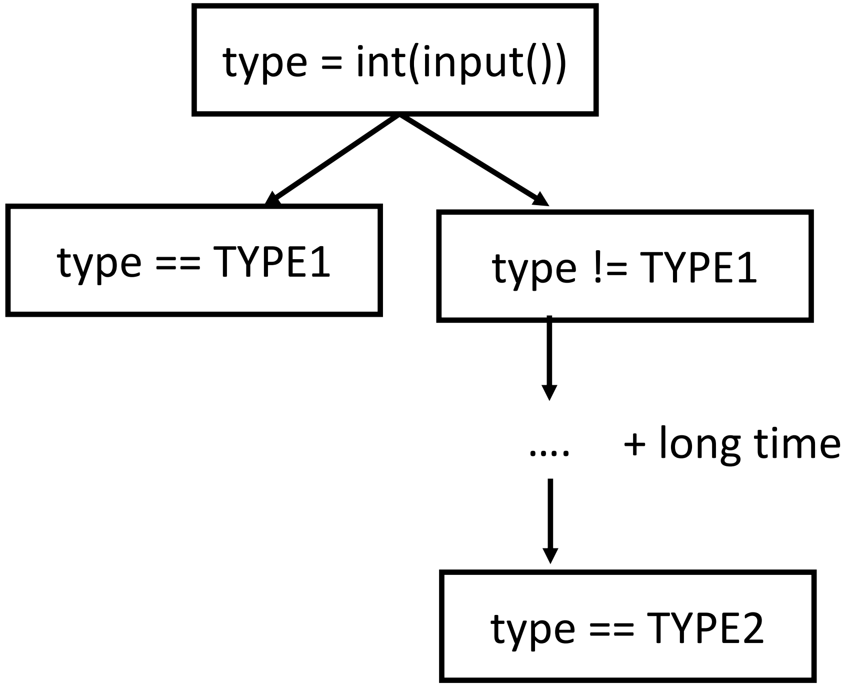
```
type = int(input())

if type == TYPE1:
   parse_TYPE1()

...

if type == TYPE2:
    parse_TYPE2()
```



Problem3: Over-constrained paths results in no test cases

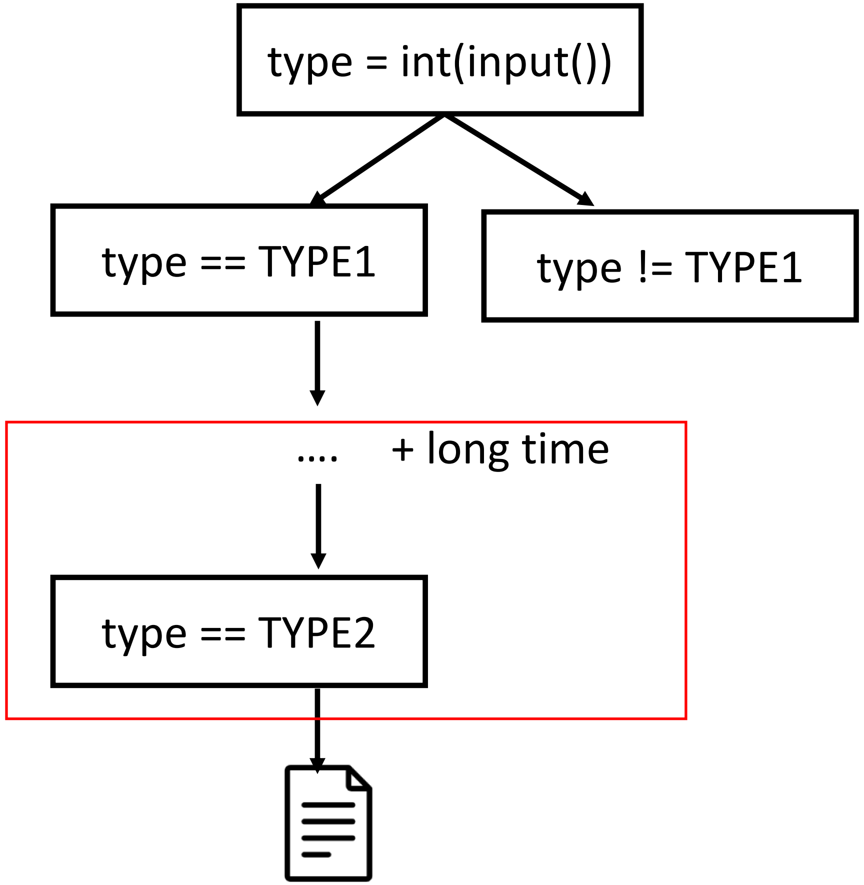
```
type = int(input())

if type == TYPE1:
  parse_TYPE1()

...

if type == TYPE2:
   parse_TYPE2()
```

If these branches are independent



Solution3: Solve constraints optimistically

```
type = int(input())

if type == TYPE1:
   parse_TYPE1()

...

if type == TYPE2:
    parse_TYPE2()
```



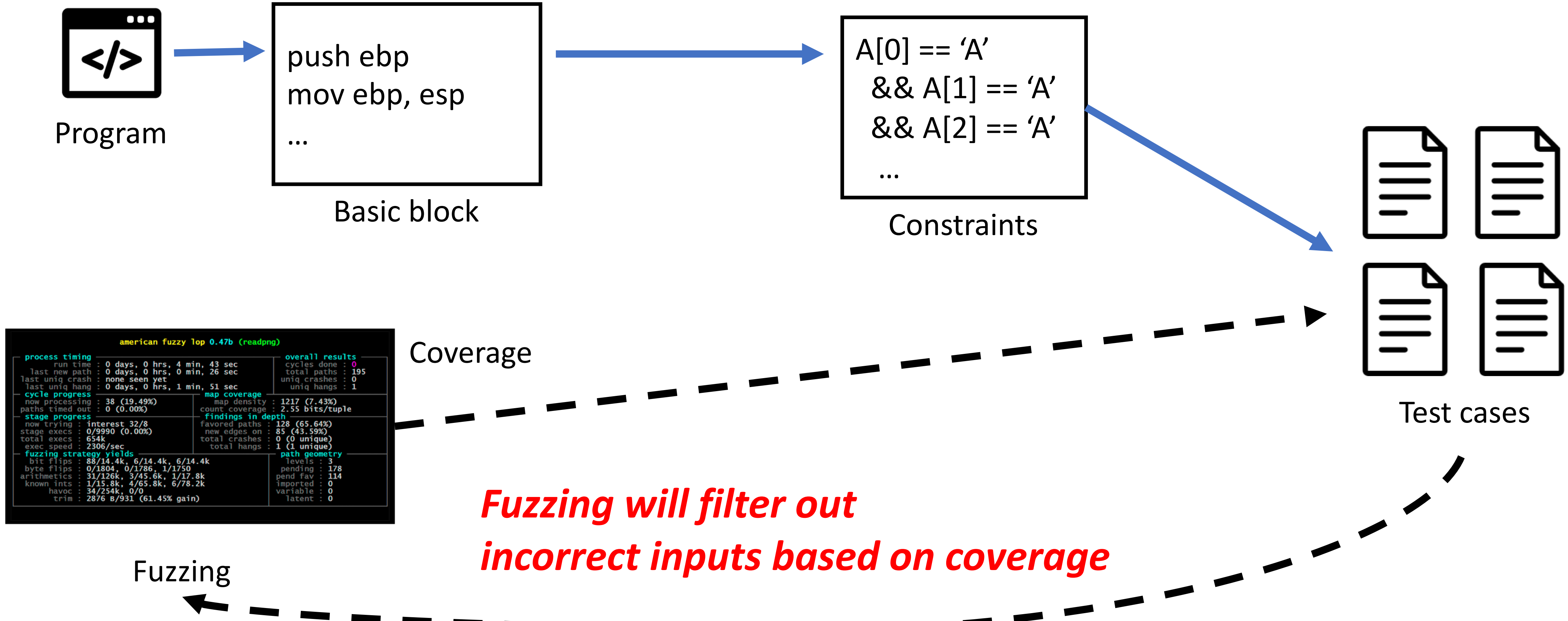
Our decision: Solve only the last constraint in the path

```
type = int(input())

if type == TYPE1:
  parse_TYPE1()

…

if type == TYPE2:
   parse_TYPE2()
```

- Simple: Only one constraint
- High chance to pass the branch

- Only waste a small solving time

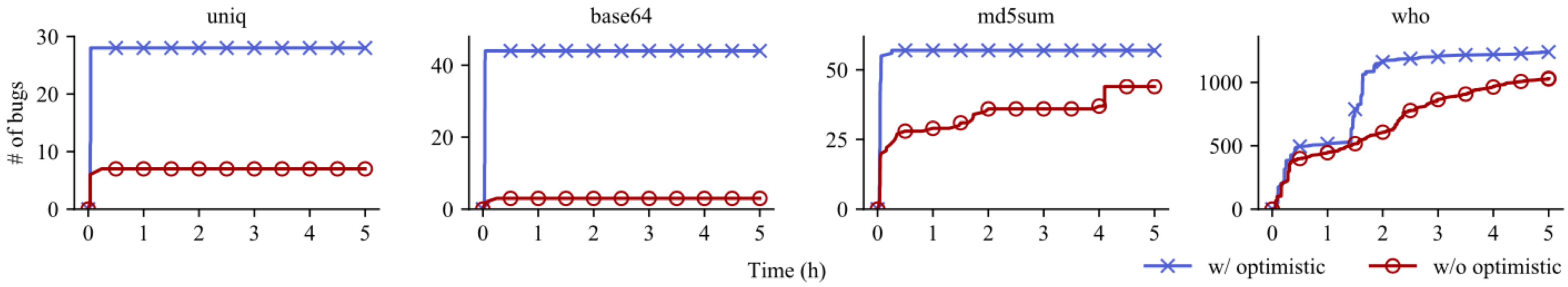
In hybrid fuzzing, generating incorrect inputs are fine due to fuzzing



Fuzzing will filter out incorrect inputs based on coverage

Optimistic solving helps to find more bugs

- LAVA-M dataset



Implementation

- 16K LoC of C++

- Intel Pin: emulation
- Z3: constraint solving

- Will be available at https://github.com/sslab-gatech/qsym

Evaluation questions

- Scaling to real-world software?

- How good is QSYM compared to
  - Driller (a state-of-the-art hybrid fuzzing)
  - Vuzzer (a state-of-the-art fuzzing)
  - Fuzzing and symbolic execution

QSYM scales to real-world software

- 13 bugs in real-world software

Program	CVE	Bug Type	Fuzzer
lepton	CVE-2017-8891	Out-of-bounds read	AFL
openjpeg	CVE-2017-12878	Heap overflow	OSS-Fuzz
	Fixed by other patch	NULL dereference	
tcpdump	CVE-2017-11543*	Heap overflow	AFL
file	CVE-2017-1000249*	Stack overflow	OSS-Fuzz
libarchive	Wait for patch	NULL dereference	OSS-Fuzz
audiofile	CVE-2017-6836	Heap overflow	AFL
	Wait for patch	Heap overflow × 3	
	Wait for patch	Memory leak	
ffmpeg	CVE-2017-17081	Out-of-bounds read	OSS-Fuzz
objdump	CVE-2017-17080	Out-of-bounds read	AFL

QSYM can generate test cases that fuzzing is hard to find

- e.g.) ffmpeg: Not reachable by fuzzing

```
if( ((ox^(ox+dxw))
            | (ox^(ox+dxh))
            | (ox^(ox+dxw+ dxh))
            | (oy^(oy+dyw))
            | (oy^(oy+dyh))
            | (oy^(oy+dyw+ dyh))) >> (16 + shift)
            || (dxx | dxy | dyx | dyy) & 15
            || (need_emu && (h > MAX_H || stride > MAX_STRIDE)))
{ ... return; }
// the bug is here
```

Compare QSYM with Driller, a state-of-the-art hybrid fuzzing

- Dataset: 126 binaries from CGC

- Run only *one* instance of concolic execution for ***5 min***
  - i.e., remove fuzzing
- Compare code coverage

QSYM achieved more code coverage than Driller in most cases of CGC

- Among 126 challenges
  - QSYM achieved more: 104 challenges
  - Driller achieved more: 18 challenges

QSYM achieved more code coverage due to its better performance

- e.g., CROMU_00001
- To achieve new code coverage, seven stages are required
  - Add one user → Add another user → login → send to message → …
- QSYM can reach the stage, but Driller cannot in time

Driller achieved more code coverage if nested branches exist

- Driller can find inputs for nested branches by a single execution due to forking
- QSYM requires re-execution
  - NOTE: Our experiment allows only one instance of concolic execution

QSYM outperforms other techniques in LAVA-M dataset

- LAVA-M dataset: inject hard-to-find bugs in real-world software
- 5 hour run

	uniq	**base64**	**md5sum**	**who**
Total	28	44	57	2,136
FUZZER	7 (25 %)	7 (16 %)	2 (4 %)	0 (0 %)
SES	0 (0 %)	9 (21 %)	0 (0 %)	18 (1 %)
VUzzer	27 (96 %)	1 (2 %)	0 (0 %)	23 (1 %)
QSYM	28 (100 %)	44 (100 %)	57 (100 %)	1,238 (58 %)

Discussions & Limitation

- Use of less accurate test cases
  - Requires efficient validators
  - e.g., exploit generation

- Implementation status
  - Only support x86, x86_64
  - No floating point support

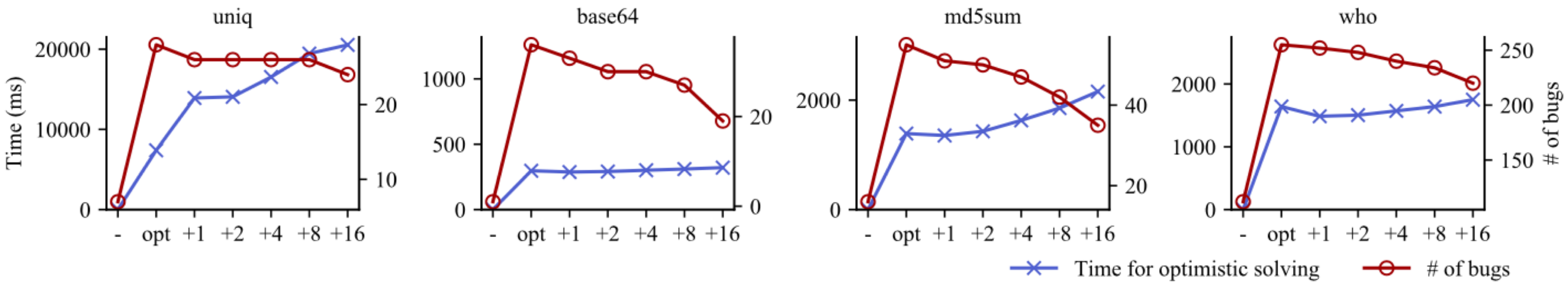
Conclusion

- Hybrid fuzzing scalable to real-world software
  - 13 bugs in real-world software
- Outperform a state-of-the-art hybrid fuzzing and other bug finding

- https://github.com/sslab-gatech/qsym

Thank you

Using only the last constraint is good for time and bug finding



Number of instructions that are not emulated by QSYM due to its limitation

Challenge	Not emulated	Total
NRFIN_00026	4 (0.02 %)	24,315
NRFIN_00032	4 (0.00 %)	4,784,433
CROMU_00016	18 (0.06 %)	31,988
KPRCA_00045	25 (0.00 %)	81,920,092
KPRCA_00009	27 (0.23 %)	11,512
NRFIN_00027	178 (0.73 %)	24,449
CROMU_00028	1,154 (0.01 %)	18,626,977
CROMU_00010	1,467 (0.18 %)	811,819
CROMU_00020	3,492 (11.15 %)	31,306
KPRCA_00013	4,589 (0.02 %)	18,746,620
CROMU_00002	14,977 (3.92 %)	381,793
NRFIN_00021	18,821 (33.26 %)	56,583
KPRCA_00029	31,800 (0.16 %)	19,604,258

- 13 / 126 challenges: At least one
- 3 / 126 challenges: More than 1% of total instructions